Docket No.: 1450.1035

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re the Application of:

Toshiaki SARUWATARI, et al.

Serial No. 10/686,638 Group Art Unit: 2181

Confirmation No. 1129

Filed: October 17, 2003 Examiner: Vincent Lai

For: INFORMATION PROCESSING UNIT AND INFORMATION PROCESSING METHOD

PETITION FOR WITHDRAWAL OF FINALITY OF OFFICE ACTION

Director of Technology Center 2100
PO Box 1450
Alexandria, VA 22313-1450

Sir:

This is a response asserting that the outstanding Office Action mailed March 19, 2007 the finality of the same are improper. The outstanding Office Action has a period for response set to expire on June 19, 2007.

Overview

The Applicant submitted a Request for Continued Examination with the previous Amendment filed on January 3, 2007. The Examiner indicated in the outstanding Office Action that the application is eligible for continued examination, and the finality of the previous Action was accordingly withdrawn pursuant to 37 C.F.R. § 1.114. In the Amendment, the Applicant added new claim 20, which recited features somewhat similar to method claim 19, but written in means-plus-function format. The Applicant *particularly* used means-plus-function language in claim 20 *requiring* interpretation under 35 U.S.C. § 112 ¶ 6. Despite the fact that means-plus-function claims must be analyzed differently than method claims per 35 U.S.C. § 112 ¶ 6, the Examiner improperly made the current Office Action final. The Examiner has not indicated how the features of means-plus-function claim 20 are being considered, whether interpretation under 35 U.S.C. § 112 ¶ 6 has *not* been taken, or what corresponding parts of the specification correlated to the recited "means" in claim 20.

Additionally, in the outstanding Office Action, the Examiner failed to consider references from the IDS of July 31, 2006 that were properly submitted to and received by the PTO. In the current Office Action, the Examiner stated that "copies of the non-patent literature (items AM, AN and AO) were not submitted and thus not considered" (see page 2, item 4, of the Office Action).

As indicated on the Return Postcards (see attached copy), the USPTO acknowledged receipt of the documents, which were included with the IDS. As such, the record is incomplete and entry of a final Action is improper.

The Examination of Means-Plus-Function Claims Differs from Method Claims

The Applicant respectfully submits that the outstanding Office Action is improper and the finality of the same is improper for the following reasons. A means-plus-function claim must be examined differently than method claims pursuant to 35 U.S.C. § 112 ¶ 6, which states "such claim *shall* [emphasis added] be construed to cover the corresponding structure, material, or acts described in the specification and equivalents thereof." As such, a means-plus-function claim containing somewhat similar limitations to a method claim presents new issues for examination.

Further, the Applicant respectfully points out that the present final Office Action does not meet the requirements of 37 CFR 1.113(b), which states "[i]n making such final rejection, the examiner shall repeat or state all grounds of rejection then considered applicable to the claims in the application, clearly stating the reasons in support thereof." In the present Action, the Examiner repeated the rejections made in the method claim, without more. In fact, it is readily apparent from the Office Action that the Examiner merely cut his arguments with respect to claim 19 and pasted them into the rejection of claim 20 (compare the rejection of claim 19 on pages 8 and 9 with the rejection of claim 20 on pages 9 and 10). There is no evidence in the record that the Examiner ever analyzed claim 20 in the manner required by 35 U.S.C. § 112 ¶ 6.

As there were no means-plus-function claims in the claim set prior to the addition of claim 20, it is clear that no such analysis was conducted for prior Amendments. Contrary to MPEP 707.07(f), there is no clear explanation anywhere in the present Office Action that the Examiner has properly considered means-plus-function claim 20. According to 37 CFR 1.104(b), the Examiner's answer must be complete as to all matters. Thus, the finality of the present Office Action was improper.

References not Considered with the July 31, 2006 IDS

Additionally, references AM, AN and AO properly filed with the IDS were not considered. These references were properly filed and received by the USPTO, as indicated by the Return Postcard. Without consideration of these documents, the record is incomplete. Hence, the finality of the outstanding Office Action is further improper for this reason.

For convenience, further copies of the references are attached hereto.

Conclusion

In light of the above, the outstanding Office Action and the finality thereof are improper. Therefore, it is respectfully requested that the finality of the Action be withdrawn.

Respectfully submitted,

STAAS & HALSEY LLP

Date: June 13, 2007

By: /J. Randall Beckers/
J. Randall Beckers
Registration No. 30,358

1201 New York Ave, N.W., 7th Floor
Washington, D.C. 20005
Telephone: (202) 434-1500
Facsimile: (202) 434-1501

Please Date Stamp and return

IDS, Form PTO 1449 & Copies of References Cited Therein, attachment 1(g), Check $180.00 ✓

APPLICANT(S): Toshiaki SARUWATARI, et al.

SERIAL NO: 10/686,638

CONFIRMATION NO. 1129

TITLE: INFORMATION PROCESSING UNIT AND INFORMATION PROCESSING METHOD

FILING DATE: October 17, 2003

DOCKET NO: 1450.1035/DMP:MPS:mbs

DUE DATE: August 17, 2006

Please Date Stamp and return

IDS, Form PTO 1449 & Copies of References Cited Therein, attachment 1(g), Check $180.00 ✓

APPLICANT(S): Toshiaki SARUWATARI, et al.

SERIAL NO: 10/686,638

CONFIRMATION NO. 1129

TITLE: INFORMATION PROCESSING UNIT AND INFORMATION PROCESSING METHOD

FILING DATE: October 17, 2003

DOCKET NO: 1450.1035/DMP:MPS:mbs

DUE DATE: August 17, 2006

17

2334c IEICE Transactions on Electronics
E76-C(1993)July, No.7, Tokyo, JP
1993

1080 p.1080-1092

G06F9/38E

PAPER Special Issue on New Architecture LSIs

Speculative Execution and Reducing Branch Penalty on a Superscalar Processor

Hideki ANDO†, *Member*, Chikako NAKANISHI†, *Nonmember*, Hirohisa MACHIDA†, *Member*, Tetsuya HARA†, *Nonmember and* Masao NAKAYA†, *Member*

SUMMARY Superscalar processors improve performance by exploiting instruction-level parallelism (ILP). ILP in a basic block is, however, not sufficient on non-numerical applications for gaining substantial speedup. Instructions across branches are required to be executed in parallel to dramatically improve performance. That is, speculative execution is strongly required. Boosting is a general solution to achieving speculative execution. Boosting labels an instruction to be speculatively executed, and the hardware handles side-effects. This paper describes the efficient implementation of boosting in terms of cost/performance trade-offs. Our policy in implementation is beneficial in code scheduling heuristics, penalties imposed by code duplication to maintain program semantics, and area cost. This paper also describes a branch scheme which minimizes branch penalty. Branch delay causes crucial penalties on the performance of superscalar processors since multiple delay slots exist even in a single delay cycle. Our scheme is the fetching of both sequential and target instructions, and either of them is selected on a branch. No delay cycle can be imposed. This scheme is realized by a combination of static code movement and hardware support. As a result, we reduce branch penalty with small cost. Simulation results show that our ideas are highly effective in improving the performance of a superscalar processor.
key words: superscalar, VLIW, speculative execution

1. Introduction

Multiple instruction execution is a key aspect in improving the performance of microprocessors. Most state-of-the-art microprocessors[1]–[4] have superscalar architectures to exploit instruction-level parallelism (ILP).

ILP is constrained primarily by two factors: data dependence and control dependence. Data dependence is classified into true dependence, anti-dependence, or output dependence. Instructions with true dependence must be executed sequentially. On the other hand, anti- or output dependence can be removed because they are artificially introduced. For example, register renaming makes these instructions executed in parallel. Unfortunately, true dependence is dominant. In particular, the number of instructions in a basic block on nonnumerical applications is small (approximately five), and most of them must be executed sequentially due to true dependence.

Manuscript received December 25, 1992.
Manuscript revised March 9, 1993.
† The authors are with LSI Laboratory, Mitsubishi Electric Corporation, Itami-shi, 664 Japan.

Control dependence is imposed due to conditional branches. Since it is unknown whether instructions in a basic block after a conditional branch will be executed, instructions in the basic block before the branch and instructions after the branch must be executed sequentially.

ILP which can be exploited within basic blocks is limited on non-numerical applications. It is only 1.5 under an ideal assumption which includes infinite hardware.[5] In fact, those current superscalar microprocessors which exploit ILP only in a basic block show small performance benefit over scalar processors on non-numerical applications.

Limit studies[5]–[8] show us the limit of ILP which can be exploited under ideal assumption (e.g. infinite number of function units, one cycle operation, etc.). Although these studies show different number of limits (2–40) because of different assumptions, they indicate that *speculative execution* is particularly important in gaining ILP benefits. Speculative execution is defined as the execution of instructions whose ultimate validity depends on the condition of a branch. Speculative execution, therefore, minimizes control dependence.

Compiler techniques[9]–[12] exist to realize speculative execution. Compiler approaches have large scope and good heuristics in instruction movement, and do not have run-time overhead. Unfortunately, basically these techniques move instructions across branches only if the operation of the moved instructions neither changes the program semantics nor causes an exception. We term those operations *safe* and *legal*. A speculative operation is said to be safe if that operation cannot cause an exception to occur, and a speculative operation is said to be legal if that operation does not overwrite a location whose previous value is needed by some other instructions when the program control is taken in ways other to the way the instruction of that operation is moved. Because of the limited capability of speculative movement, these compiler techniques do not yield substantial performance gain.

Loop-level optimization techniques, such as *loop unrolling* and *software pipelining*,[13] are also useful for exploiting ILP. While these techniques are particularly useful on numerical applications because inner-most loops consume a large part of CPU time, they are

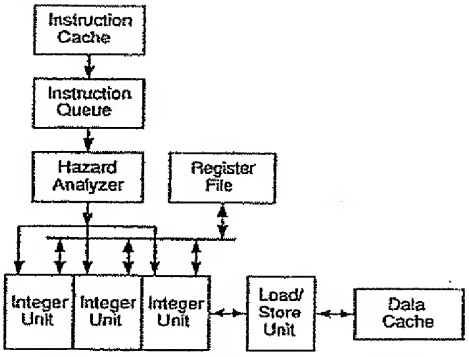


Fig. 1 Hardware organization of SARCH.

IF : Instruction fetch
ID : Instruction decode, hazard analysis, branch, and instruction issue
EXC : Execution
MEM : Data cache access
WR : Register write

Fig. 2 Pipeline.

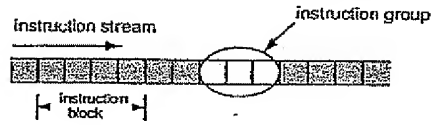


Fig. 3 Instruction group.

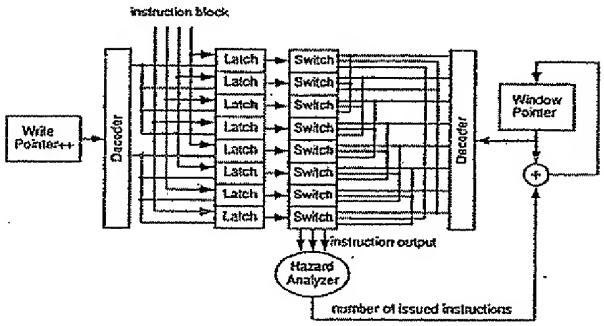


Fig. 4 Instruction queue.

tions sequentially. The decision of the number of integer ALUs will be discussed in Sect. 5.

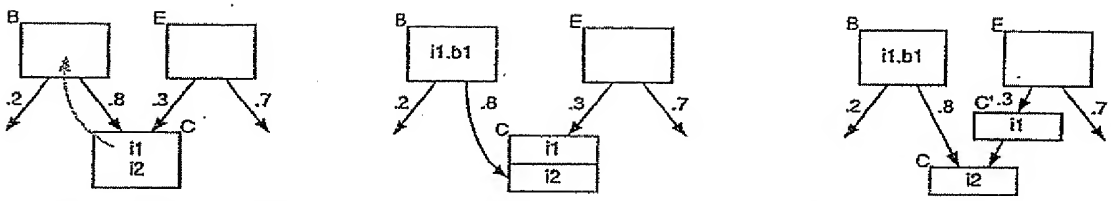
Figure 2 shows the pipeline—*IF*: instruction fetch, *ID*: instruction decode, hazard analysis, execution of a branch instruction (including branch address calculation), and instruction issue, *EXC*: execution and data address calculation, *MEM*: data cache access, and *WB*: register write. The instruction set is nearly identical to the MIPS R2000 RISC instruction set.[27] The latency of all integer instructions except load and branch instructions is one cycle. Since data is loaded from data cache in the MEM stage, the latency of a load instruction is two cycles. A branch instruction is executed in the ID stage. Thus, the latency of the branch instruction is two cycles. SARCH uses neither a delayed load nor a simple delayed branch like R2000 does.

In general, the amount of ILP changes dynamically on run-time. Issuing fixed number of instructions to sustain peak ILP like very long instruction word (VLIW) machines yields an enormous number of no-ops in the code. Therefore, dynamic hazard analysis is required to keep code size in scalar machines. The hazard analyzer determines which instruction can be issued and then only instructions without hazards are issued. In other words, the hazard analyzer inserts no-ops dynamically.

Hazards can be caused by data dependence and resource conflicts. Data dependence analysis is performed by the comparison of register numbers and checking a reservation table of the registers. The comparison among destination register numbers and source register numbers of candidates for parallel issue is sufficient for the hazard analysis of instructions with one cycle latency because all results before the WB stage are bypassed. On the other hand, those candidates that will use data loaded from the data cache should be blocked until the data becomes available because the latency of a load instruction is two cycles. A register reservation table is used for an availability check of loaded data. The register reservation table has 32 entries associated with a register number. An issued load instruction marks the entry associated with its destination register number, and a load instruction which completes the execution resets the entry associated with its destination register number. By checking the register reservation table, the hazard analyzer can find the availability of source registers.

SARCH fetches four instructions every cycle. We call those instructions the *instruction block*. To pack instructions, we should allow an instruction group, instructions which can be issued in parallel, to be located across an instruction block boundary (Fig. 3). To meet this requirement, we employ a dynamic window. The window moves along the instruction stream. The instructions in the window are supplied to the hazard analyzer and do not depend on instruction block boundaries. The window moves according the number of instructions issued every cycle.

Figure 4 shows the implementation of the dynamic window. The dynamic window is realized by the instruction queue. The queue has eight latches which store instructions. Four fetched instructions are written in the latches 0–3 or 4–7 every cycle. Each latch is connected to three bit lines through a switch box. The switch box is controlled by the window pointer indicating the top entry of the window. Data in three successive latches indicated by the window pointer is read out to the bit lines. For example, if the window pointer is 2, data in latch 2, 3, and 4 are read out. The movement of the window is performed by



(a) Desired code movement. (b) Schedule when C is not a fall-through block of B. (c) Schedule when C is a fall-through block of B.

Fig. 7 Code duplication on boosting limited in a trace.

transferred to D from A (40%), a code movement from C to A seems to create better code. We should, however, consider the penalty imposed on off-trace paths. Since C is a join block, that is, C has a predecessor (block E) other than B, code duplication to E is required to maintain semantics when a code movement from C to A is performed. This duplication may impose a penalty on the off-trace path. If B is also a join block, the scheduler needs code duplication again for the predecessors of B. In general, the amount of duplicated code increases with the number of conditional branches that are moved across. Thus, a code movement which replies on static branch prediction does not necessarily create the best code in terms of the performance.

The capability of code movement along an unlikely path is also effective to suppress penalties imposed on unlikely paths when code duplication is necessary. The boosting model which allows code movement only in a trace has constraint on code duplication. Consider the case in Fig. 6 again. When a code movement from C to A is performed, the code should be duplicated to E. Since this code duplication is speculative, boosting may be required (Notice that labeling of boosting is required only in the case of unsafe or illegal movement). In the case in Fig. 6, however, since the path from E to C is not a probable path, and consequently, is not a trace, code movement from C to E is not allowed if it is an unsafe or illegal movement.

The scheduler that relies on trace paths has three options to handle in this case. The first option is that those movements where duplication is unsafe or illegal are not allowed. Although this option is easily implemented, it imposes great constraints on scheduling, and consequently degrades the performance dramatically. The second option is to change the entry point of the branch in B into the point of the next instruction to the moved instruction if C is not a fall-through block of B (Fig. 7(b)). This is effective, but it cannot be performed if C is a fall-through block of B. The third option is to dynamically make a new basic block (block C') which contains duplicated code (Fig. 7 (c)). This option keeps the semantics, but may cause a penalty on the off-trace path (E to C through C').

In contrast, the scheduler in two-way boosting has

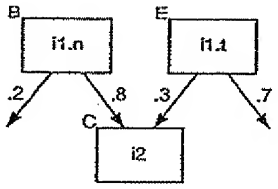


Fig. 8 Code duplication on two-way boosting.

no constraints on code duplication. Two-way boosting allows the duplicated instruction to be moved into E in all cases. Figure 8 shows the duplication for the duplication in the case of Fig. 7(c). If the scheduler moves i1 into B, i1 is labeled i1.n, and duplicated code is placed in E labeled i1.t, where the label .n means that a instruction labeled .n is valid if the associated branch is not taken, and similarly the label .t means that an instruction labeled .t is valid if the associated branch is taken. The duplicated instructions will be probably issued along with instructions which belong to the original block. Therefore, penalties imposed by code duplication can be minimized. In our code scheduler, the second option described above and two-way boosting is incorporated to produce the best schedule.

Another advantage of two-way boosting is removing unconditional branch instructions for a short branch-and-join. Consider the example shown in Fig. 9(a). As shown in the control flow graph Fig. 9(b), the block A branches to B and C, but they are joined into D with a short run. The MIPS R2000 optimized code and the SARCH optimized code is shown in Fig. 9(c) and (d), respectively. Notice that the unconditional branch instruction (jump EXIT) is removed. This optimization is similar with one in the guarded instruction model.[20] A guarded instruction is conditionally executed depending on a value in the register designated in the code. Two-way boosting, however, has an advantage over the guarded instruction model since it does not need predicate dependence. That is, in the guarded instruction model, B and C can be issued in parallel, but A, B and C cannot be executed in parallel since the execution of B and C should wait until the branch condition is determined in A.

If a branch instruction is executed and the branch is taken, target instructions which were statically moved are executed in the next cycle. Because of the sequential placement of the target instructions, no branch delay is imposed. At the same time, a new target address (not the original target address, but the newly created target address after the movement of the target instructions) is output to the instruction cache. The dynamically fetched target instructions are appended below the statically moved target instructions in the instruction queue. Since these instructions can be executed immediately, the pipeline does not stall. Figure 11 shows the image of the instruction queue after fetch of the new target instructions.

Figure 12 is an example explaining the branch scheme. Figure 12(a) is an example code in memory where i2–i4 are instructions to be executed when the branch is untaken, and t1–t9 are instructions to be executed when the branch is taken. Figure 12(b) is the branch taken case. The instruction group is a group of instructions which are issued in parallel. This grouping is done by the hazard analyzer described in Sect. 2. Recall that the instructions in the *window* (three instruction wide) pointed by the *window pointer* (denoted by *wp* in the figure) are read out to the hazard analyzer. In the cycle n, the instructions (i1, branch, t1) are issued due to no hazard in this example. Because the branch is taken, t1 is issued along with i1 and branch. In the next cycle, the cycle $n+1$, (t2, t3, t4) are issued without delay because of the static movement of the target instructions. At the same time, the target PC (address of LAB) is sent to the instruction cache. In the next cycle, the cycle $n+2$, (t5, t6, t7) can be issued immediately because t6–t9 are loaded to the queue in this cycle†.

Figure 12(c) shows the case of the untaken branch without delay. In the cycle n, (i1, branch, t1) is read out from the instruction queue, and only (i1, branch) is issued. The instruction t1 is nullified in this cycle because it is the target instruction. In the next cycle, the window pointer jumps and is set to 8 so that (i2, i3, i4) can be read from the instruction queue. These instructions are issued in the cycle $n+1$ without delay.

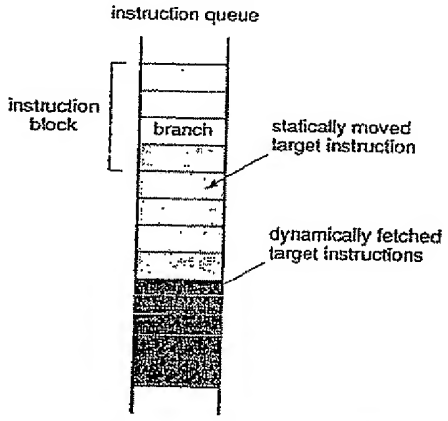
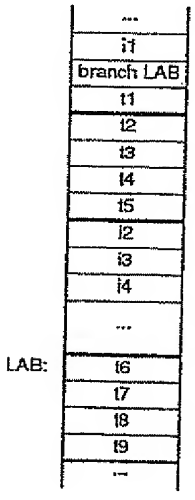
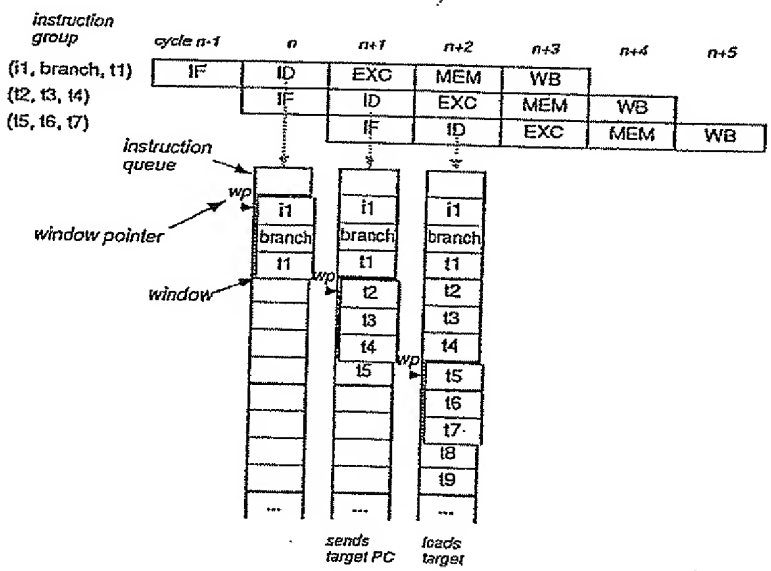


Fig. 11 Image of the instruction queue after fetch of the new target instructions.

† The figure presents that (t5, t6, t7) are fetched in the cycle $n+1$, but if we precisely present, t5 is fetched in the cycle n due to the instruction prefetching, while (t6, t7) are fetched in the cycle $n+1$.



(a) Code example.



(b) Taken branch.

Fig. 12 Pipeline on branch execution.

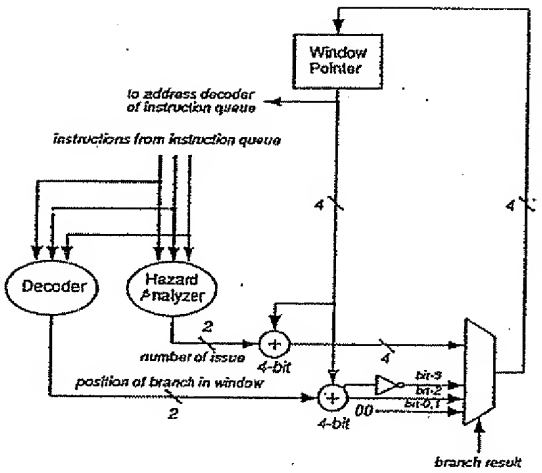


Fig. 13 Window pointer control.

(Fig. 12(d)). The average issue rate is between two and three, and thus the possibility of the pipeline stall due to the untaken branch is reduced. Notice that the bandwidth of the instruction fetch (four-instruction per cycle) is determined to meet the requirement for sustaining the peak issue rate (three-instruction per cycle), not to meet the requirement for the branch scheme. That is, we do not pay cost for implementation of the branch scheme in the instruction fetch bandwidth.

4.2 Comparison with a Previous Branch Scheme Prefetching from Both Directions

Prefetching from both directions is employed in mainframes (e.g. IBM370/168). In these machines, the condition code (CC) which is needed for the branch is determined in a *late* stage. Because of late CC setting, as well as higher fetch rate than the issue rate, the machine can afford to fetch instructions from both directions. There are two buffers to store instructions for the sequential and target path before an instruction register. When an instruction decoder following the instruction register finds a branch, the target address, if operands for target address calculation are available, is sent to the instruction cache, and the target is fetched in the target buffer. If the branch was resolved, either the sequential or target instruction from the buffers is selected according to the result.

Our scheme is similar to the scheme above, but is efficiently realized in the RISC-type superscalar machine. Before describing the primary benefits, the secondary benefit is that our scheme does not send the target address until the branch is found to be taken. In the mainframe schemes, the target address is sent to the instruction cache before the target is not found to be needed. Since the target address is not the sequential address of the previous instruction address in general, this extra instruction reference does not take advantage of spacial locality. Although the LSI technology has advanced, the primary cache which can be integrated in a single chip is limited (currently from 8 K[3] to 20 KB[1]). Therefore, the extra instruction reference might cause the cache (and TLB) miss in a small processor unlike mainframes. The handling of these misses should be suppressed because it is unknown whether the processor should really handle them or not. This control might be realized, but makes the hardware complicated.

Ignoring the secondary benefit, it is useless to employ the branch scheme in mainframes as it is because CC is determined *early* in RISC machines. RISC machines do not need memory-operands for an ALU operation; ALU operations are performed between only register-operands. Therefore, a RISC machine has neither a operand-address calculation stage nor a memory-operand fetch stage before an execution stage, which are required for mainframe instruction-set architecture. That is, the execution stage (or EXC in SARCH, see Fig. 2) is placed immediately after the register-operand fetch stage (or ID in SARCH). Therefore, CC is available one cycle after the issue of the CC-set instruction. Furthermore, *compare-and-branch*, which is employed in SARCH, does not need CC setting since the equivalent operation is performed in the ID stage together with CC testing and branch address calculation. In other word, the branch instruction can be executed immediately after it is fetched, while the execution is suspended until CC is set by the previous instruction in mainframes. Therefore, the branch scheme where the target address is sent if the decoder finds the branch instruction is useless in RISC machines.

The revised version to the branch scheme in mainframes is to decode instructions early before they are decoded in the instruction decoder. This predecoding enables to send the target address and fetch the target before the branch execution, but requires registers to hold the fetched instruction block other than the buffers and a decoder to find the branch instructions; the original instruction decoders are still needed to determine the branch direction. This duplication costs approximately 3K transistors ((400 transistors (registers) + 300 transistors (decoder)) × 4 (instructions per block) + 250 transistors (selector to share the branch address generator)). Unlike this revised scheme, our scheme does not need the hardware for predecoding.

Finally, we should consider a penalty when the prefetch from both directions is not successfully performed. In SARCH code, basic block length, which is cycles consumed from the entrance to the exit, is extremely short. According to dynamic statistics we collected from benchmark programs (see Table 1), 37% of the basic blocks is just a single cycle in length; 49% is less than two cycles in length. Under this situation,

which can be exploited in a basic block is strictly limited. As mentioned in Sect. 1, most instructions in non-numerical applications are dependent with true dependence. Therefore, there is little ILP in the basic block. Further-more, basic block scheduling does not solve the problem of control dependence. As a result, the performance is not improved significantly by basic block scheduling.

Figure 15 shows performance improvement when two-way boosting and our branch scheme are introduced. The average speedup is 1.36x in the machine with two ALUs, and is 1.71x in the machine with three ALUs. These numbers in improvement indicate that up to three ALUs are beneficial in our scheduling scheme. The machine with four ALUs, however, achieves only a 1.2% performance benefit over the machine with three ALUs. The primary reason of the performance limit is that the load/store instructions must be executed sequentially, while the ALU instructions can be executed in parallel. Therefore, the parallel execution of load/store instructions is the next most important aspect to improving performance.

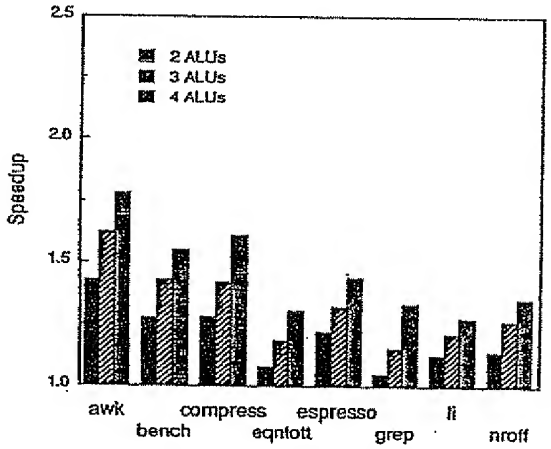


Fig. 14 Performance improvement with basic block scheduling only.

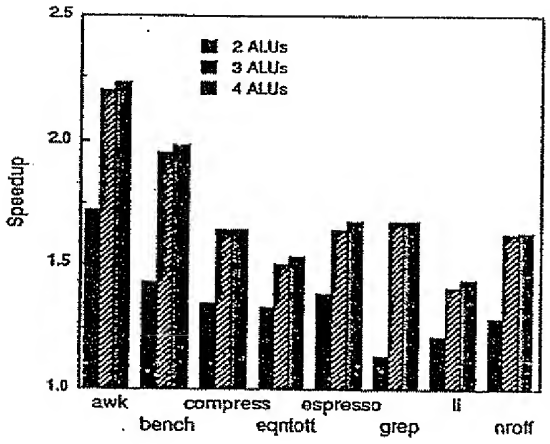


Fig. 15 Performance improvement with two-way boosting and our branch scheme.

Figure 16 summarizes the performance benefits which are obtained by two-way boosting and our branch scheme in the three issue machine. In model A, only basic block scheduling is performed. In model B, the scheduler schedules code with two-way boosting, but our branch scheme is not introduced. Model C employs both two-way boosting and our branch scheme. The figure shows that two-way boosting achieves a 21.4% performance improvement over basic block scheduling, and our branch scheme achieves a further 8.1% performance improvement. The total performance improvement of 29.5% is achieved. Thus our ideas are highly effective in performance improvement.

An interesting question to ask is comparison with the boosting which schedules instructions across multiple branches in terms of performance and hardware amount. This type of boosting is supported with shadow structures for speculative state buffering. From the view point of the hardware amount, it is obvious that our boosting needs a smaller amount of hardware because shadow structures are not needed. For example, the number of transistors in a six-read, three-write register file, which is required for three-issue SARCH, is 23 K transistors†, and thus extra 23 K transistors are required for a shadow register file. Only increase from one-level original boosting is extra squashing logic in pipelines, but it is extremely trivial. Yet, this increase is equivalent or smaller than two- or more-level original boosting.

To compare performance with the original boost-

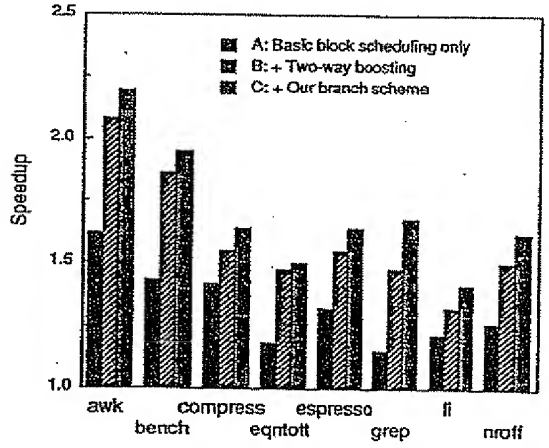


Fig. 16 Performance impact of two-way boosting and our branch scheme.

† The amount of transistors is quite significant because it is 27% more than the number of transistors in a single integer unit in SARCH.

substantial speedup gain.

Acknowledgments

We would like to thank H. Komiya and Y. Horiba for giving us opportunity to work on this project.

References

(1) Blanck, G. and Krueger, S., "The SuperSPARC Microprocessor," In *Proc. Compcon'92*, pp. 136–141, Feb. 1992.
(2) Diefendorff, K. and Allen, M., "The Motorola 88110 Superscalar RISC Microprocessor," In *Proc. Compcon'92*, pp. 157–162, Feb. 1992.
(3) Dobberpuhl, D. W., Witek, R. T., Allmon, R., Anglin, R., Bertucci, D., Britton, S., Chao. L., Conrad, R. A., Dever, D. E., Gieseke, B., Hassoun, S. M. N., Hoeppner, G. W., Kuchler, K., Ladd, M., Leary, B. M., Kadden, L., MacLellan, E. J., Meyer, D. R., Montanaro, J., Priore, D. A., Rajagopalan, V., Samudrala, S. and Santhanam, S., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE J. Solid-State Circuits*, vol. 27, no. 11, Nov. 1992.
(4) Groves, R. D. and Oehler, R., "An IBM Second Generation RISC Processor Architecture," in *Proc. Compcon'90*, pp. 166–172, Feb. 1990.
(5) Wall, D. W., "Limits of Instruction-Level Parallelism," in *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, Apr. 1991.
(6) Jouppi, N. P. and Wall, D. W., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, Apr. 1989.
(7) Lam, M. S. and Wilson, R. P., "Limits of Control Flow on Parallelism," in *Proc. 19th Int. Symp. on Computer Architecture*, pp. 46–57, Jun. 1992.
(8) Smith, M. D., Johnson, M. and Horowitz, M. A., "Limits on Multiple Instruction Issue," in *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, Apr. 1989.
(9) Bernstein, D. and Rodeh, M., "Global Instruction Scheduling for Superscalar Machines," in *Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pp. 241–255, Jun. 1991.
(10) Ellis, J. R., "Bulldog: A Compiler for VLIW Architecture," *Ph. D. Thesis*, Yale U/DCS/RR-364, Yale University, Feb. 1985.
(11) Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, vol. C-30, no. 7, pp. 478–490, Jul. 1981.
(12) Nicolau, A., "Percolation Scheduling: A Parallel Compilation Technique," *Computer Sciences Technical Report 85-678*, Cornel University, May 1985.
(13) Lam, M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proc. ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pp. 318–328, Jun. 1988.
(14) Murakami, K., Irie, N., Kuga, M. and Tomita, S., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," in *Proc. 16th Int. Symp. on Computer Architecture*, pp. 78–85, Jun. 1989.
(15) Tomasulo, R. M., "An efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, vol. 11, no. 1, pp. 25–33, Jan. 1967.
(16) Hwu, W. W. and Patt, Y. N., "Checkpoint Repair for Out-of-Order Execution Machines," in *Proc. 14th Int. Symp. on Computer Architecture*, pp. 18–26, Jun. 1987.
(17) Smith, J. E. and Pleszkun, A. R., "Implementation of Precise Interrupts in Pipelined Processors," in *Proc. 12th Int. Symp. on Computer Architecture*, pp. 36–44, Jun. 1985.
(18) Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J. and Hwu, W. W., "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in *Proc. 18th Int. Symp. on Computer Architecture*, pp. 266–275, May 1991.
(19) Colwel, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B. and Rodman, P. K., "A VLIW Architecture for a Trace Scheduling Compiler," in *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, Apr. 1987.
(20) Hsu, P. Y. T. and Davidson, E. S., "Highly Concurrent Scalar Processing," in *Proc. 13th Int. Symp. on Computer Architecture*, pp. 386–395, Jun. 1986.
(21) Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rau, B. R. and Schlansker, M. S., "Sentinel Scheduling for VLIW and Superscalar Processors," in *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, Oct. 1992.
(22) Smith, M. D., Lam, M. S. and Horowitz, M. A., "Boosting Beyond Static Scheduling in a Superscalar Processor," in *Proc. 17th Int. Symp. on Computer Architecture*, pp. 344–355, May 1990.
(23) Smith, M. D., Horowitz, M. A. and Lam, M. S., "Efficient Superscalar Performance Through Boosting," in *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 248–259, Oct. 1992.
(24) Hennessy, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann Publisher, Inc., 1990.
(25) Lee, J. K. F. and Smith, A. J., "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.
(26) Phillips, M. J., "Performance Issue for the 88110 RISC Microprocessor," in *Proc. Compcon'92*, pp. 163–168, Feb. 1992.
(27) Kane, G., *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.
(28) McFarling, S. and Hennessy, J. L., "Reducing the Cost of Branches," in *Proc. 13th Int. Symp. on Computer Architecture*, pp. 396–403, Jun. 1990.
(29) Smith, M. D., "Support for Speculative Execution in High-Performance Processors," *Technical Report CSL-TR-93-556*, Stanford University, Jan. 1993.

8207 IEI Micro
10(1990)February, No.1, Los Alamitos, CA,

FEATURE

Implementing Sparc in ECL

Two companies successfully joined forces to design a small, low-cost system capable of large mainframe performance.

Emil W. Brown
Anant Agrawal
Trevor Creary
Michael F. Klein
David Murata
Joseph Petolino

Sun Microsystems Inc.

In 1987, as the first Sun 4 workstation moved into full production, an emerging emitter coupled logic technology caught Sun Microsystems' attention. ECL promised very large scale integrated densities at much higher operating frequencies than CMOS (complementary metal-oxide semiconductor) technology. The convergence of this technology with our Scalable Processor Architecture (Sparc) would for the first time enable a complete microprocessor to be implemented in ECL.

Sun initiated a joint development project with Bipolar Integrated Technology to implement Sparc using BIT's new bipolar process. Initial work indicated that a clock cycle time of 12.5 nanoseconds, an execution rate of 1.3 clock cycles per instruction, and benchmark performance of 60 million instructions per second and 12 million floating-point operations per second were achievable with an entry-level price of $100,000.

This was an aggressive goal considering that mini-supercomputers then under development targeted clock rates of 25 to 40 megahertz and $200,000 to $500,000 entry-level prices. We were confident that the simplicity of Sparc would put the processor core on less than a dozen chips, and Sun's workstation heritage would fit the entire 80-MHz processor onto one circuit card. We minimized the cost by using air cooling, pin grid array and dual in-line packaging, 10K technology, and conventional printed circuit boards.

We believe that early adoption of new technology is the key to creating competitive products. However, chip development can no longer be separated from system design since much of the system now resides on the silicon itself. With these factors in mind, we assembled a design team that consisted of an equal number of IC engineers and system or board-level designers. We knew that board-level issues such as RAM access characteristics, transmission line design, and clock skew control as well as system architecture would determine many of the requirements for the VLSI chips.

Here, we briefly review both ECL technology and the features of BIT's ECL technique and discuss how board and cache considerations influenced the chip designs. Discussion of the integer unit pipeline, system interface signals, and coprocessor interface concludes the article. The chip set, now completed, sells commercially from BIT as the B5000 series. See The B5000 Microprocessor box.

Technology

ECL is a digital bipolar technology generally used for applications in which cost and power dissipation are less important than switching speed. The relatively large bipolar transistors of a traditional ECL design resulted in circuits with lower integration density than their CMOS counterparts. In addition, the power dissipation in ECL was high because the gates had to be biased to drive longer, more capacitive internal signal lines.

0272-1732/90/0200-0010$01.00 © 1990 IEEE

8207 IE. Micro
10(1990)February, No.1, Los Alamitos, CA,

Sparc in ECL

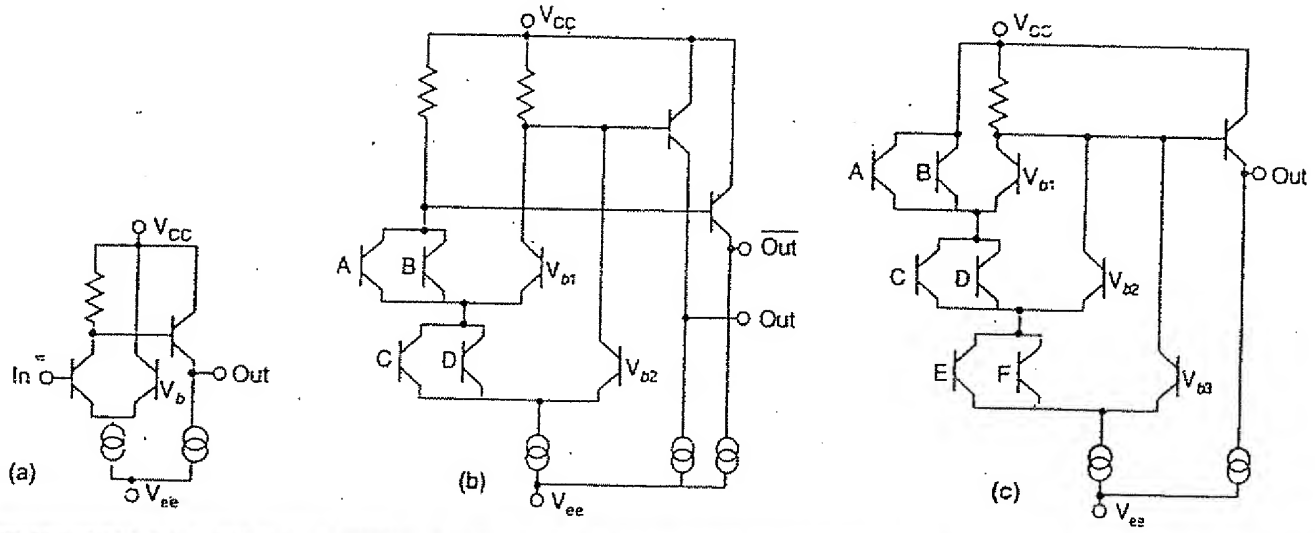


Figure 1. An ECL inverter (a), a two-level series gate implementing the function $(A + B)(C + D)$, and its complement (b), and a three-level series gate implementing the function $(A + B)(C + D)(E + F)$.

BIT achieved a double breakthrough when it

- reduced the physical size of the bipolar transistors, and
- provided a typical unloaded gate propagation delay of 375 picoseconds, while biasing each gate with 70 microamperes of static current. More traditional ECL techniques use 200 μA to 1 mA to achieve comparable switching speed.

With three layers of interconnect metallization, the process is ideally suited for building VLSI devices.[1,2]

All three layers of metal distribute power on these devices to minimize the voltage drops along the bus and to avoid metal migration. A package with an embedded copper-tungsten slug having high thermal conductivity dissipates the power. The die bonds directly to the slug, which transfers the heat to the top of the package. A heat sink in a forced airstream dissipates the heat. The resulting thermal resistance (θ_{μ}) is about 2.5 degrees centigrade per watt.

An ECL inverter consists of a differential pair, a current source, a load resistor, and an output driver (Figure 1a). By adding transistors in parallel to the input transistor of the differential pair, we create an Or function. If we add another differential pair between the original pair and the load resistor, we create an And function.

This stacking of differential pairs is called series-gating. A traditional ECL process allows only two levels of series-gating. For example, the logic function $(A + B)(C + D)$ can be implemented as one gate (Figure 1b). The BIT process allows three levels of series-gating, which supports functions such as $(A + B)(C + D)(E + F)$, as shown in Figure 1c. The penalty for the additional functionality is an increase in the propagation delay; however, this increase is generally less than if the function was decomposed into two gates. Three levels of series-gating proved useful in constructing the shift registers that make up the diagnostic scan chain in the integer unit, or IU, and the floating-point controller, or FPC. The required multiplexer and latch functions combine into one gate.

ECL circuits can efficiently drive transmission lines at high frequencies. So the IU and the FPC each contain a set of low-impedance (25-ohm) output drivers to drive system buses directly. We minimized the timing skew between these outputs by matching delays on the chip. We also designed package traces from the die to the pin to match the impedance of the drivers.

BIT technology supports two ECL interface standards, designated 10K and 100K. The 100K interface uses temperature compensation circuitry to minimize the temperature-induced shift in the switching threshold. The 10K interface specifies the amount of threshold shift that is allowed across the operating temperature range. All 100K circuits operate with a −4.5 volt supply, while 10K circuits operate at −5.2 volts. We chose to adhere to the 10K standard because it offers a wider selection of standard components. The cooling system maintains a maximum junction-temperature gradient of 25°C across the board to minimize differences in switching thresholds between devices. This system is necessary to maintain noise immunity and to control temperature-induced switching skew.

Board design influences

We started with several studies of cache designs. Although we looked at a variety of concepts, the simplest design offered the shortest access time, and the more complex designs did not offer comparable advantages.

8207 IEEE Micro
10(1990)February, No.1, Los Alamitos, CA.

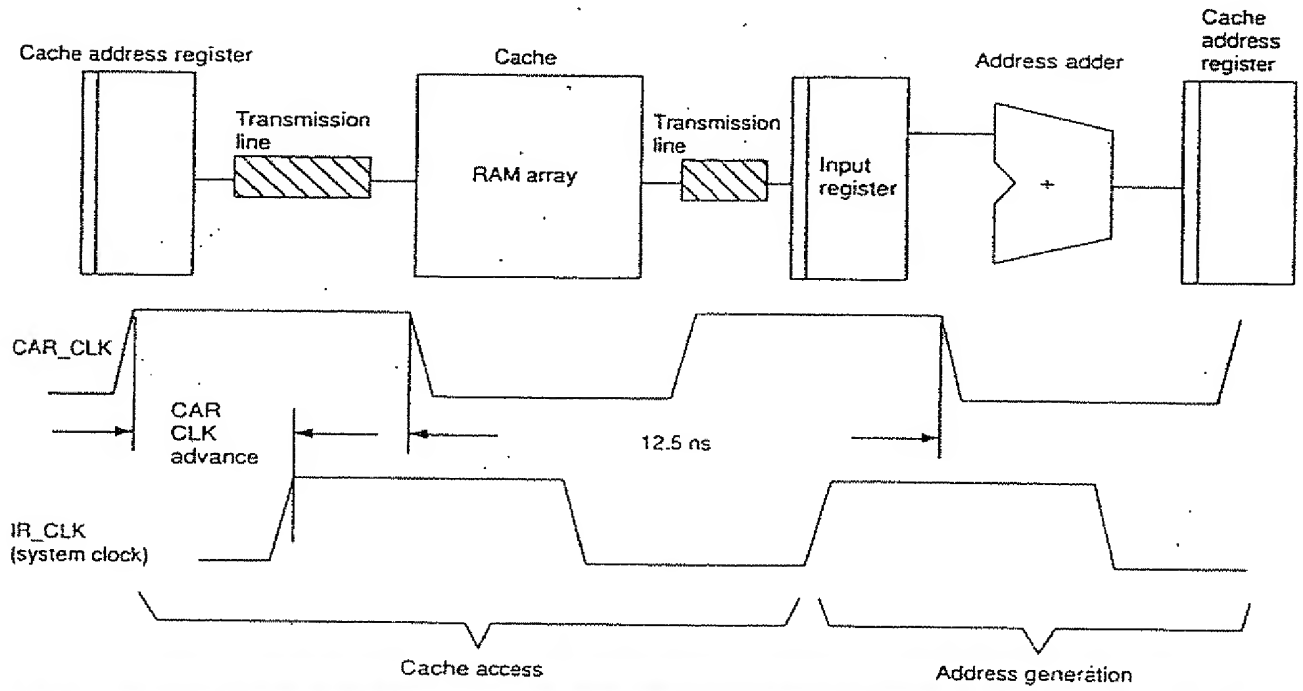


Figure 3. Address generation.

several basic decisions we made to minimize signal propagation delays:

- only the IU drives the cache address bus;
- the cache data bus only connects to the IU and the floating-point unit, or FPU; and
- the cache write operation does not limit the cycle time; it is preferable to use two cycles for each write.

As a result, all data entering the IU or FPU comes from the cache RAMs, and cache data must pass through the IU or the FPU to get to memory. Noncached data (such as system control registers, or power-on bootstrap code) moves through the cache. First, the cache controller saves the addressed word from the cache in a temporary register, writes the desired data into the vacated cache location, and from there into the IU or FPU, then restores the original cache data from the temporary register.

The CPU core seen in Figure 2 contains the Sparc integer unit, the five-chip FPU, the cache RAM array, the tag match chip, and four copies of the system data path gate array. The FPU consists of the controller, two register file chips, the double-precision ALU, and the double-precision multiplier. The tag match chip implements the cache miss logic and the tag portion of a four-entry translation (lookaside buffer) cache, or TLB. The system data path contains the data portion of the TLB and provides interconnections between the major units of the CPU via the CPU bus. It also contains the memory bus interface.

Cache design

With each address pin of a RAM presenting 5 to 7 picofarads of loading in DIP form, the AC impedance of an address bus with the RAMs packed as closely as possible is about 25 ohms. With 25 RAMs (18 for data and parity, and 7 for tags), the delay on the address transmission line totals about 10 ns. By splitting the cache into two banks, each driven by its own copy of the address, we halve the delay time to 5 ns. We provided the IU with 25-ohm differential drivers for 16 address lines, enough to support a 512-Kbyte cache. A further split into four banks is possible by using external drivers, but these can add significant delay and skew.

We optimized the tag-access and compare operations for speed in several ways. First, the tag RAMs are four times smaller than the data RAMs and thus have a shorter access time. Second, we designed a high-speed ECL gate array (the tag match chip) to perform the tag match computation, and third, placed the tag RAMs so that they are the first to receive the address from the IU.

Systems with one-cycle access caches have in the past required RAMs with a read-access delay significantly shorter than the cycle time of the machine. We assumed that such RAMs would be unavailable, too expensive, or too small for this machine, so we designed the pipeline to allow extra time in the cache access stage. We "borrow" time from the address generation stage (see Figure 3), which operates in less than 7.5 ns. An on-chip cache address register (CAR) is clocked with an early clock to enable the next cache

8207 IEEE Micro
10(1990)February, No.1, Los Alamitos, CA,

dress changes, and the store data arrives one cycle later than the address. There must also be time to stop the write pulse from occurring if the tag check results in a cache miss. If the store instruction is followed by a nonmemory-access instruction, its execution will be overlapped with the third cycle of the store, reducing the effective cost of the store to two cycles.

The timing of store and swap (seen in Figure 4a) illustrates the critical placement of the write pulse between the arrival of the data and the end of the address. Integer load-double (Figure 4b) takes two separate load cycles. Integer store-double (Figure 4c) is the only three-cycle instruction. Floating-point store and store-double instructions take one cycle in the pipeline, although it takes three cycles to complete the cache write (Figure 4d).

IU pipeline

Sparc offers a simple instruction set based on a register-to-register paradigm. Only load and store instructions access memory and the only addressing modes used by these instructions are register-plus-register and register-plus-immediate. Register specifiers appear in the same bit fields of every instruction. Branch instructions carry an immediate, program counter-relative offset.[5]

The simplicity of Sparc allows all of the instructions to fit neatly into a fixed pipeline which, on the B5000, consists of five stages: fetch (F), read (R), execute (E), memory (M), and write (W). (Note in The Sparc Architecture box that pipelining is not a required feature of any Sparc implementation.) Each stage completes its processing in one clock cycle. During the F stage instructions move from memory into the processor. Then the processor reads operands from the register file, decodes opcodes, and detects instruction dependencies in the R stage. The operands enter either the arithmetic and logic unit (ALU) or the shift unit in the E stage. Load and store instructions in the M stage fetch data operands from memory, and arithmetic instructions use it to move the arithmetic result back across the chip to the write port of the register file. In the W stage the processor writes the ALU result or the data from memory into the register file.

In the next clock cycle we use standard result-forwarding techniques to keep the pipeline full, even when an instruction's result is used.[6] We added a new data path to this implementation to allow load instructions to execute in one cycle, or two cycles when the next instruction requires the data being loaded. The hardware detects and handles the two-cycle case; compiler support is not required. We call this an interlock action and in general use it when an instruction encounters a resource conflict or data dependency and cannot be issued in the current cycle.

Figure 4 illustrates most of the multiple-cycle instructions. The integer store and store-double instructions require two and three cycles respectively because the IU register file does not have a third read port to access the stored data in the R stage. However, the floating-point load, load-double, store, and store-double instructions take just one cycle in the pipeline due to external 64-bit buses and the separate floating-point register file.

Instruction queue

The B5000 contains a 64-bit data input bus on which two instructions can be fetched in parallel. One or both of these instructions will be inserted into a queue, which has a maximum depth of four instructions. The queue allows the pipeline to complete one instruction every cycle even when memory access instructions occasionally use the data bus (Figure 5). The B stage of

continued on p. 19

The Sparc Architecture

The reduced instruction-set computer, or RISC, architectures developed at the University of California at Berkeley form the basis of the Scalable Processor Architecture. Sparc contains only 32-bit-long instructions in three formats. Operand specifiers appear in fixed positions in the instructions to enable rapid register file access. Delayed control transfer instructions allow an instruction that follows a control transfer instruction to execute before the transfer of control occurs.

In Figure B we can see that Format 1 is used only by the subroutine Call instruction. This format has a 30-bit displacement, which allows a Call to any word-aligned address in the virtual address space. Format 2 supports the Sethi instruction and the Branch instructions. The 22-bit displacement allows Branch instructions to span 16 Mbytes of the virtual address space. Format 3 supports all of the other instructions. It has three 5-bit operand specifiers, or two 5-bit operand specifiers and one 13-bit signed immediate constant. The three-

8207 I. E Micro
10(1990)February, No.1, Los Alamitos, CA

Table A.
Three of the four general types of the Sparc instruction set.

Instruction	Data types	Data width	Variations	Notes
Memory operations				
Load	Signed	Byte, halfword	Alternate space[1]	—
Store	Unsigned	Word	Floating-point queue	—
		Double word	Floating-point status reg.	—
		Single, double	—	—
Swap		Word	Alternate space[1]	Atomic[2]
Load/Store	Unsigned	Byte	Alternate space[1]	Atomic[2]
Integer computational				
And, Or	—	Word	Not[3]	—
XOR	—	—	Set cc[4]	
Add	—	Word	Set cc,[4] tagged[5]	—
Sub	—	—	Extended	—
MULSCC	Signed	Word	—	Set cc[4]
Save	—	Word	—	Performs Add
Restore	—	Word	—	—
Shift	—	Word	Left, right	Shift by 0 to 31 bits
Read	—	Word	PSR,[6] WIM[7]	—
Write	—	—	TBR,[8] Y[9]	—
Control transfer				
Branch	Signed	22-bit displacement	Integer cc[4]	—
—	—	—	Floating-point cc[4]	—
—	—	—	Coprocessor cc[4]	—
—	—	—	Execute delay[10]	—
—	—	—	Annul delay[10] if not taken	—
Call	Signed	30-bit displacement	—	Delayed
Jump and Link	—	Word	—	Delayed
Return from Trap	—	Word	—	Privileged, delayed

[1]Access to one of 255 privileged, alternate address spaces
[2]Load and Store occurs indivisibly in memory.
[3]The second operand is logically inverted.
[4]Condition codes
[5]The least significant two bits of the data act as simple type tags.
[6]Processor status register
[7]Window invalid mask
[8]Trap base register
[9]Y register (used by MULSCC as an accumulator)
[10]The instruction immediately following a control transfer instruction

Byte	8 bits
Halfword	16 bits
Word	32 bits
Double word	64 bits
Single	32-bit floating-point value
Double	64-bit floating-point value

In Tables A and B we list each individual instruction and its variations. Note that some implied variations don't exist. For example, Sparc contains no instructions for signed store, alternate space floating-point load or store, shift left arithmetic, or floating-point convert to self.

While pipelining is not a part of Sparc, the instruction set design allows efficient pipelined implementations. By keeping the instructions simple, it is relatively easy to overlap the execution of several instructions at once.

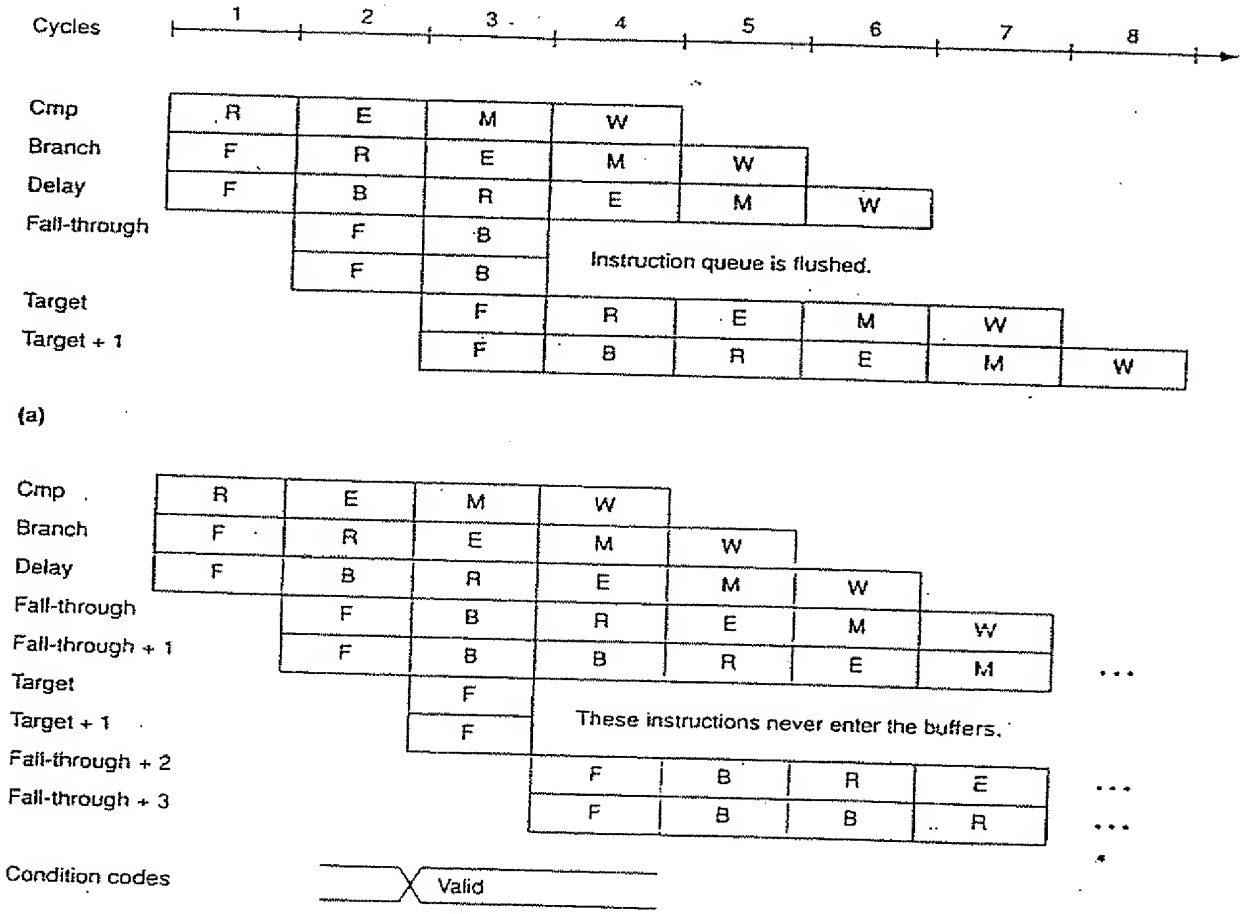


Figure 6. Branches: taken (a) and not taken (b).

fetched. The branch resided at an even-word address, so its delay instruction was also fetched at the same time. Complexities arise when the fall-through or delay instructions are not in the instruction queue. These infrequent cases do not significantly affect performance, although they do complicate the logic design.

The instruction at the target address of a branch is always fetched in the E stage of the branch (Figure 6). In the subsequent cycle 3, when the compare (Cmp) instruction computes and makes known the condition codes, the target is either decoded immediately or discarded. If the branch is not taken, the prefetched instructions following the branch/delay pair (the fall-through instructions) execute, so no cycles are lost.

FPU, coprocessor interfaces

The B5000 implements the Sparc floating-point interface and coprocessor interface symmetrically, so discussion of the floating-point interface applies equally to the coprocessor. Sparc allows register-to-register floating-point operations to execute and complete in the "background" while the pipeline continues to execute integer instructions. However, a floating-point operation may complete by generating an exception, rather than an arithmetic result.

To pinpoint the instruction causing an exception, the FPC maintains a queue of pending instructions and their addresses. The queue can be read after an exception occurs to determine which instruction actually caused the exception and which subsequent instructions had been issued but not completed. Each entry of the queue contains an instruction and its address. The IU dispatches instructions on the store-data bus in the R stage of the pipeline. The addresses are generated on the FPC chip, which has a copy of the E stage program counter (called the XPC). The IU manages the XPC with its "increment XPC" control, and by loading it from the store-data bus following any control transfer.

8345 Computer Architecture News
19(1991)April, No.2, New York, US

E

G06F9/38F1

Reducing the Branch Penalty by Rearranging Instructions in a Double-Width Memory

Manolis Katevenis† and Nestoras Tzartzanis†
Computer Science Institute, FORTH
Heraklio, Crete, Greece

P.15-27

ABSTRACT: In a pipelined processor with an instruction-fetch throughput of two (consecutive) instructions per cycle, one method to reduce the branch penalty is to rearrange the code by placing (copies of) instructions from *both* targets of a branch in the double-width fetch stream after that branch. This scheme is of interest e.g. when the number of fetch cycles is large, thus making it hard to fill all the delay slots with instructions from before the branch, and when the hardware has super-scalar capabilities but the compiler does not find enough instructions for parallel execution in the basic block where a branch is predicted to go. We study this scheme of rearranging instructions, and we evaluate its performance (execution time and code size) in the case where *no* parallel instructions are scheduled in the delay slots.

KEYWORDS: *Pipelined computer architecture, Branch penalty, Delayed branch, Delay slot, Rearranging instructions into delay slots, Super-scalar computer architecture, Double-width instruction memory.*

1. INTRODUCTION

Branch and jump instructions disrupt the regular flow of instructions through the pipeline of modern processors, thus negatively affecting their performance. In order to cope with this situation, numerous schemes have been deviced and implemented to reduce that loss in performance [LeSm84] [Lilj88] [FaHe86] [DeRo87]. Figure 1 presents a model of a pipeline that we will use. In that figure, the first instruction in the pipe, which was fetched from address A, is a conditional branch to address B. Say that the pipeline has k instruction-fetch stages, that branch instructions compute (or know) their target address by the end of the $(m+1)$th stage, and that the truth or falsity of the branch condition is known by the end of the $(n+1)$th stage. In a naive architecture and implementation, the n cycles after the branch instruction are lost because it is not yet known which instruction to process (assuming $n \geq m$).

In a slightly more clever implementation, these cycles are lost only when the branch succeeds, by letting the instructions that were fetched in the meanwhile from $A+1$ through $A+n$, execute and commit in case the branch condition fails. Unfortunately, the more frequent case is for the branch to succeed, especially when *un*conditional jumps, procedure calls, and returns are also counted. The situation can be ameliorated using various hardware or software methods. By feeding every fetch address A into a special hardware *branch-target buffer* (BTB) (version #1), the target address B may be generated in less than $m+1$ cycles, and we can start processing instructions from that address right away. If the BTB works in one pipe stage, there are no cycles lost whenever we hit in the BTB and the prediction of the branch direction that the BTB makes is correct. Another kind of branch-target buffer (version #2) is a cache that contains the first few instructions of basic blocks that are destinations of branches (or jumps/calls/returns); (a loop buffer is another similar piece of hardware). If that cache can be accessed in less than k cycles, then it can quickly supply the first instructions after a branch, while the main instruction-fetch pipeline gets restarted. A third hardware method of speeding up branches can be applied to pipelines where the branch target address becomes known (significantly) earlier than the branch condition ($m < n$, or presence of BTB #1, or a special *prepare-to-branch* instruction placed by the compiler a few slots before the branch itself). By fetching instructions simultaneously from the *two* different possible streams – $A+i$ and $B+j$, the hardware can later decide which one of these streams to execute from.

On the other hand, the software methods to reduce the cost of branches are all based on letting the n delay-slot instructions, $A+1$ through $A+n$ do useful work (*delayed branches*). One scheme lets these instructions unconditionally commit, and expects the compiler to move there useful instructions that can safely be moved; these instructions may originate from before the branch, if they do not affect it, or from one of the two targets, if they are harmless should the branch go in the other direction. Another scheme lets the delay-slot instructions commit only if the branch goes in a prespecified direction – otherwise, they are *squashed*; this lets the compiler move into the delay slots instructions from one of the two targets, regardless of what their effect would be should the branch go in the other direction. The success

† Also with the Department of Computer Science, University of Crete, Heraklio, Crete, Greece.
Postal Address: Computer Science Institute, Foundation of Research and Technology – Hellas (FORTH), P.O.Box 1385, Heraklio, Crete, 711-10 Greece.
E-mail Address: katevenis@csi.forth.gr
Telephone: +30 (81) 229302. Fax: +30 (81) 229342.

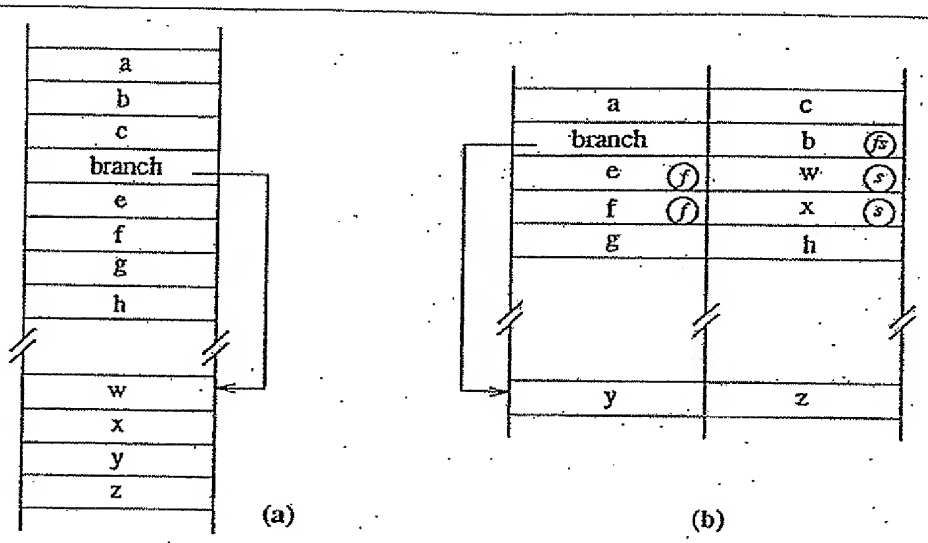


Figure 2: Rearranging instructions around a conditional branch

In (a) instructions are conventionally arranged in an instruction memory. In (b) the width of the memory was doubled, and the instructions were rearranged (instruction addresses increase from left to right and from top to bottom). The branch instruction is followed by n=3 delay slots (single or double). The instructions in these slots that are marked "*f*" are executed only if the branch fails, those marked "*s*" are executed only if the branch succeeds, while those marked "*fs*" are executed in either case. Instruction *b* was moved from before the branch into the first delay slot and will be executed unconditionally. Before fetching has had the time to revert to *(y, z)* (if the branch is taken), *(e, w)* and *(f, x)* are automatically fetched and the CPU can selectively execute either *e* and *f* or *w* and *x*.

the increasing size difference between the I-cache and the register file. The cycle time of the CPU is largely dictated by the cycle time of the register-file – a RAM of size one or a few kilobits, made using high power and speed cells. On the other hand, the instruction cache tends to contain one or more hundreds of kilobits, by current and future standards. Given this large discrepancy in size, the I-fetch latency is likely to grow to two or more pipeline stages. For example, in the Prisma GaAs processor [Wils89], instruction fetching required two pipeline stages (versus half a stage for the register-file cycle-time).

Section 2 presents our scheme in more detail, explaining how the delay slots can be filled in each case. In section 3 we describe the post-processor which we implemented for rearranging assembly code for our scheme, and for collecting measurements. Section 4 presents our static and dynamic measurements regarding code size and execution cycles. Section 5 compares this scheme with branch-cost reduction schemes, and section 6 draws the conclusions. Section 2 is long because it describes in detail all the various cases of code rearrangement that may arise; readers who are less interested in these can skip over its last part and go to section 3.

2. SEMANTICS AND USE OF THE DOUBLE-WIDTH DELAY SLOTS

Several variations of the proposed scheme can be defined, depending on the exact placement and the semantics of the delay slots that follow the control-transfer instructions in the double-width instruction memory. The number of choices is even greater when this scheme is combined with a super-scalar architecture, where the pairs of instructions may be intended sometimes for sequential execution, other times for parallel execution, and other times for alternative execution depending on the outcome of a recent conditional branch.

In this paper, we assume a *scalar*, rather than super-scalar, architecture, firstly because we want to study our scheme independent of super-scalar issues, and secondly because the software that we use for rearranging the instructions and collecting measurements does not perform any dependency analysis, and thus cannot decide whether or not two instructions can run in parallel. Specifically, we assume that the CPU can decode and start executing a *single* instruction per cycle, even though it fetches pairs of instructions from the I-cache. Figure 3 illustrates the implication of this assumption in a pipeline similar to that of figure 1 executing the rearranged code of figure 2(b). The instructions that are fetched during the delay slots can be distinguished in two categories. For some of the initial ones (instruction *b* here), the decoding and the beginning of execution occurs *before* the conditional branch has decided, and thus we have no choice when starting to execute them. For the rest of the delay slots, decoding only happens after the branch has been resolved; since two instructions are fetched per cycle, we can selectively decode and execute one of them in each cycle, depending on the branch decision; we will call these *selective delay slots*. Non-selective delay slots are like conventional delay slots – the processor executes something independent of the branch direction. These will typically be filled from the basic block *before* the branch instruction, while selective delay slots are filled from the two basic blocks that follow the branch. If there are k fetch-stages, and if a branch is resolved at the end of the $(n+1)$th stage, then there are n delay slots, the last k of which are selective. Note that if an implementation has additional hardware in the first stages of its execution pipeline, so that it can start processing two instructions while later finishing up

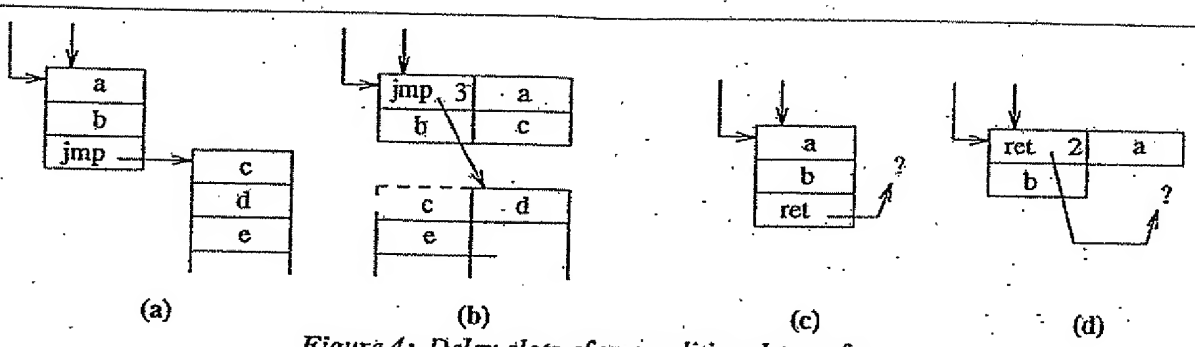


Figure 4: Delay slots of unconditional transfers

In this example, the pipeline has n=3 delay slots. The target of the *jump* instruction in (a) is known at compile time, so this code is transformed as in (b). However, the target of the *return* in (c) is not known, so the rearrangement results in (d). The *jump* in (b) specifies that it uses all 3 delay slots, while the *return* in (d) says that only 2 of the slots that follow it contain valid delay instructions. Thus, in (d), a cycle will be lost during execution, but at least no code space is wasted for a *noop*.

delay slots; no cycles are lost. This is a method to achieve object-code compatibility between implementations that use the same instruction set but have different n's; the compiler would then try to optimize for the pipeline with the maximum expected number of delay slots, and the code will run with no wasted cycles on machines with smaller n.

2.1 Handling Single Jumps or Branches

Figure 4 shows how we use the aforementioned count of actual delay slots in *unconditional* jumps. First the jump instruction is moved k places up in its basic block (BB); if the BB is small, as in (a), then instructions from the target basic block are brought to fill the rest of the delay slots (part (b) of the figure); these instructions may be removed from their original place if they cannot be reached through any other control path. However, not all jumps have a target that is known at compile time: in (c), the *return* instruction, which is a register-indirect jump in RISC architectures, was moved as far up as possible in its BB (see (d)), but the rest of its delay slots could not be filled; thus, a count of 2 rather than 3 slots is specified in that instruction. (A more sophisticated optimizer could move the *return* further up by copying and merging its BB with the BB's that lead into it, but we did not do that in our measurements). The indexed jumps resulting from *switch* (*case*) statements are an even harder case, because neither is their target known nor can they be moved up in their own BB, due to address computation dependencies.

Next, let us look carefully at the delay slots of *conditional* branches. Figure 5(a) shows again the rearranged code of figures 2(b) and 3. The branch instruction specifies the count "3" of delay slots that follow the branch. We assume that there is an agreement between the compiler and the implementation as to how many of these slots are non-selective, while the rest of them are selective – in figure 5 there is always one non-selective delay slot after every branch, and in part (a) of the figure instruction b could be moved there. For the selective delay slots, the agreement is that the instruction "on the *left*" is to be executed if the branch fails, while the one "on the *right*" is for the case of successful branch. Thus, no f, s, or fs marks are actually necessary. Remember that we assumed that only one instruction per cycle is decoded, even though two instructions are fetched; thus, the selection between instructions e and w and between f and x in the two remaining (selective) slots *cannot* be based on their content, i.e. on their f or s marks – it has to be based merely on their *positioning* in the memory. The "right-hand side" selective delay

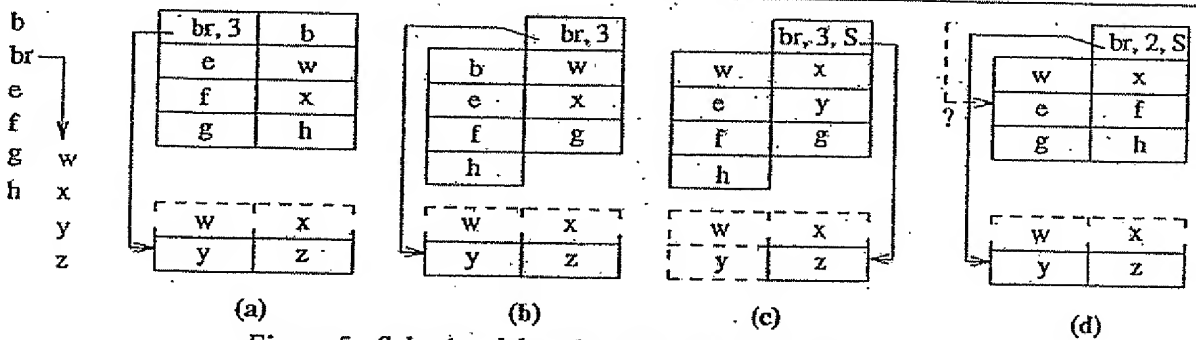


Figure 5: Selective delay slots after a conditional branch

The code of figure 2 is rearranged for up to n=3 delay slots, the first one of which is non-selective. Parts (a) and (b) show the arrangement of the instructions for the two cases of alignment of the branch. In (c), an instruction from the destination basic block had to be moved into the non-selective slot; it must be squashed if the branch fails. In (d), less instructions could be moved into the selective slots; to reflect that, the branch instruction specifies a count of only 2 delay slots.

even more if more branches are present. When the width of the instruction memory is two words, this situation cannot be handled. We then specify that the first branch will have less than the maximum number of delay slots, or we insert some *noop*(s).

Let us first consider the case when a branch is closely followed by another branch, on its *success* path (on the "right side"), as in figure 6(a). In this figure we assume that branches have n=4 delay slots, consisting of 1 non-selective and k=3 selective slots. We also assume that instructions a and p, which precede the branches, can be moved into the non-selective delay slots after them. For legibility, the right-side selective delay slots of all branches have been marked with a double line (the real implementation does not contain any such marking). In each part of this figure, original code is shown on the left and rearranged code is shown on the right. In part (a), the second branch and its non-selective delay slot (p) require a single I-fetch stream, and thus can be moved into the delay slots of the first branch. However, the selective delay slots of the second branch (*q/w, r/x, s/y*) need two I-fetch streams and thus cannot be moved there. Since only 3 of the right-side delay slots of the first branch could be filled, its slot count was reduced from 4 to 3; as a consequence, one execution cycle will be lost whenever that first branch succeeds. In order not to lose that cycle, we considered moving instructions from *one of the two* streams that follow the second branch into the selective slots of the first branch. This would be possible, but *(i)* it would further complicate the semantics of the delay slots, and *(ii)* it would require an arrangement of the instructions in the selective slots of the second branch different from the one required when control enters that block through another path (path labeled "?" in figure 6(a)). Since our measurements indicated that for $k \leq 3$ we lose at most 1% of the total execution cycles due to such reductions of the delay slots of branches, we decided to stay with the simpler scheme. Another observation is that, during the delay slot cycle which is lost for the first branch, the second branch does useful work, and thus *it* will need one less delay slot for itself; again, in the interest of simplicity of semantics, we do not take advantage of that.

Part (b) of figure 6 shows what we do when a branch is closely followed by another branch on its *failure* path (on the "left side"). Again, we cannot move more than the second branch and its non-selective slot (e) into the selective slots of the first branch. This time, however, we choose to fill the extra delay slot with a *noop* rather than reducing the number of delay slots. In this way, the *noop* cycle will be wasted whenever the first branch fails, but no cycle will be lost when that branch succeeds. If we chose to reduce the number of delay slots, then no cycle would be wasted on failure, but a cycle would be lost on success; given that branches succeed more frequently than they fail, we opted for the *noop* solution.

2.3 Other Cases Needing Attention

Figure 7 illustrates some other fine points that arise when a control-transfer instruction is so close to another that it is moved into its delay slots. In this figure, n=k, i.e. branches only have *selective* delay slots. Besides the branch-branch case, which was illustrated in fig. 6, figure 7(a) examines the jump-branch case. Here, the branch, together with as many of its (selective or non) delay slots, is moved into the jump's slots. Notice that the alignment of the *r/x* and *s/y* delay slots of the branch that is shown in figure 7(a) causes a performance loss problem: When the *jump* transfers the fetch process to instruction x, x is fetched together with q rather than together with r; thus, if the branch fails, one cycle will be lost, until r is fetched together with y. That loss is avoided when the *(q/w, r/x, s/y)* alignment is different; if no sequential path enters that block, we can always force the alignment which is favorable to us. The same performance loss problem is present for branch-branch pairs, as in figure 6(a); in that

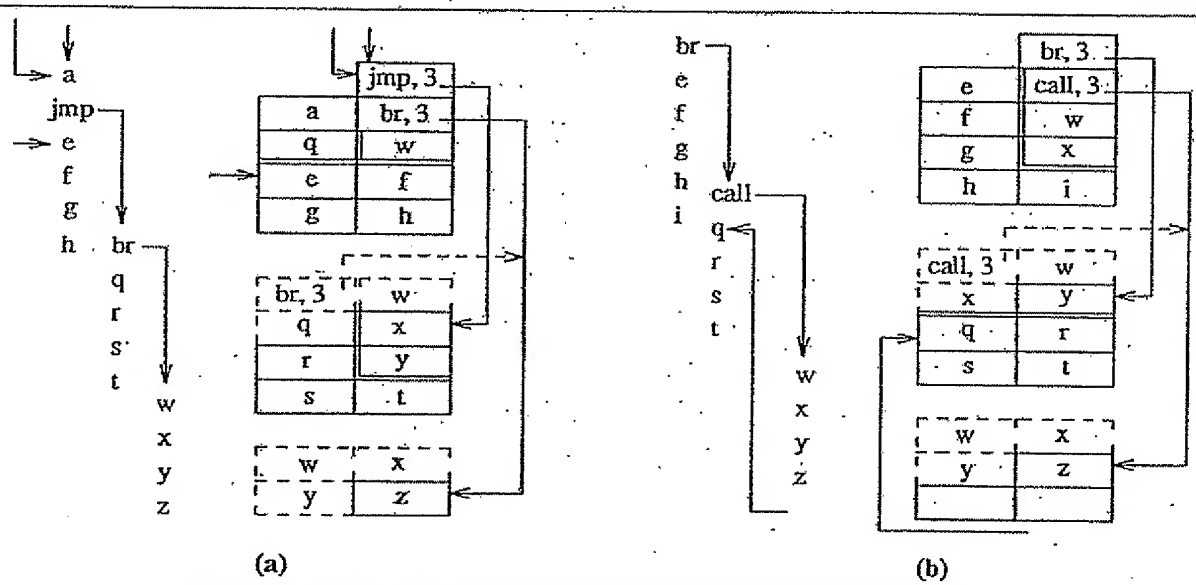


Figure 7: Other close combinations of control-transfer instructions

In this figure n=k=3. When a branch is moved into the delay slots of a jump, as in (a), the selective delay slots of the former get spread in two regions. When a procedure call is moved into the delay slots of a branch, as in (b), the return address that it saves (pointer to q) must be generated according to dynamic pipeline information rather than static instruction addresses.

We have more confidence in our measurements from the first compiler, since it fills more delay slots than the GNU compiler does, and since the MIPS compilers postpone such filling till after our postprocessor. The above compilers produced assembly code for the following four benchmark programs:

- the *cc1* part of GCC – GNU C compiler for 68020, version 1.35. The *cc1* part is the core of GCC and does the bulk of the work; it contains 83,700 lines of code; its object code (SUN cc) contains 129,129 instructions. In the dynamic measurements, *cc1* itself was fed to *gcc -O -S*; 2.6 billion instructions were executed.
- *troff*, Jan. 86 version, from the ditroff package; it contains 8,530 lines of code; its object code (SUN cc) contains 14,731 instructions. In the dynamic measurements, a 2-MByte text file was run through *troff*; 1.1 billion instructions were executed.
- *CommonTeX* – a C-language version of the TeX typesetting program; it contains 20,200 lines of code; its object code (SUN cc) contains 45,145 instructions.
- *SPICE* version 2g6, converted from FORTRAN into C by an automatic translation program. SPICE performs lots of numeric computations in order to simulate the analog behavior of electronic circuits. It contains 33,600 lines of code; its object code (SUN cc) contains 107,328 instructions.

4. CODE SIZE AND BRANCH CYCLES IN THE NEW SCHEME

Figure 8 shows the static measurements that we collected using our postprocessor, as described in the previous section, in order to evaluate the code density under the new scheme. The numbers shown are for code generated by the SUN-4 C compiler. In these measurements, wherever we show percentages in or over the "original code", we mean the code generated by the compiler and adjusted by us for an architecture with no delayed branches and no *noop*'s.

Figure 8(a) shows the static percentage of control-transfer instructions (CTI's) in the original code, for the four benchmarks, broken down in conditional branches (labeled *b*), unconditional jumps to known targets (*j*), procedure calls (*c*), returns (ranging between 0.1 and 1.7 %), and indexed jumps resulting from *switch* statements (ranging between 0.05 and 0.16 %). *Cc1*, *troff*, and *tex* have considerably more CTI's than *spice*, which is a numerical-computation program. The average basic block size ranges from 4 or 4.5 instructions for the non-numeric programs to 9 instructions for *spice*. In the MIPS code, the percentages of CTI's are in general higher – each of the branch, jump, and call category has about an additional 1%. MIPS basic blocks are about half an instruction shorter. These effects are mainly due to the existence of combined compare-and-branch instructions in MIPS. The GNU compiler produces less CTI's than the SUN/DEC compilers.

Figure 8(b) shows the value of the "replication factor". When instructions from the target of a jump or branch are moved into its delay slots, they have to be replicated when another path can still reach them at their original place; otherwise, they can simply be "moved" (figure 4(b), figure 5). The replication factor shows how often instructions have to be replicated, relative to the total number of such instruction movements. When a basic block can be reached through sequential execution, every branch or jump into it needs to replicate all instructions that it takes; when a basic block can be reached only through branches or jumps, then all but one of these CTI's need to replicate. The interesting conclusion is that this replication factor varies within a relatively narrow range around 0.77.

At this point we should note that the replication factor can be reduced with some help from the compiler. For example, the SUN-4 compiler, which we used for these measurements, compiles *while* statements into: "`if (not expr) branch to exit; loop: (body; if (expr) branch to loop); exit:`". That leads to replication of the first few instructions of the `body` after the second branch. If the *while* were compiled: "`jump to test; loop: (body; test: if (expr) branch to loop); exit:`", then no such replication would be needed.

Figure 8(c) shows the degree by which branch and jump delay slots are filled with useful instructions, as a function of k, the number of slots per CTI. These numbers refer to the right-side selective delay slots of conditional branches, and the delay slots of unconditional jumps to known targets (calls are not included – their delay slots can always be filled for the values of k that we considered). The jump delay slots are almost always filled, either with instructions from the basic block before the jump, or with instructions from its target. The branch slots can always be filled for k=1 and almost always for k=2; for k=3, 94% of them can be filled (the reasons why the rest of them cannot were illustrated in figures 5(d) and 6(a)). The product of branch instruction frequency in the original code, times k (the number of selective delay slots per branch), times the branch fullness, times the replication factor gives the amount of code size increase that is due to the selective slots of branches.

Figure 8(d) shows the overall increase of code size over the size of the original code, for each of the four benchmarks compiled by the SUN-4 compiler. Each number that is shown as a vertical bar with solid lines is broken down into its three contributions: selective delay slots of branches (labeled *s*), delay slots of unconditional jumps to known targets (*j*), and delay slots of procedure calls (*c*). (The delay slots of indexed jumps to *case*'s are not filled, and those of *return*'s are only filled with instructions from before them, so these two categories do not contribute to increasing the code size). The solid lines correspond to the cases where branches only have selective delay slots: n=k. Measurements are given for k= 1, 2, or 3 fetch pipeline stages. When n=k+1, i.e. when branches are resolved in the second execution stage and hence they have one non-selective delay slot, the contribution to increased code size by the above factors was measured to be *almost identical* – usually it increased by about 0.1%, at most by 0.4%, and in one case it decreased by 1% (remember that when n=k+1, unconditional CTI's are assumed to still have k delay slots). What changes when n=k+1 is the presence of the *non-selective* delay slots of branches, for which our postprocessor does whatever the SUN-4 compiler did. That compiler manages to fill only about 20% of them with instructions from before the branch, about 73% of them with instructions from the branch target (to be squashed (*annulled*) if the branch fails), and fills the remaining 7% with *noop*'s. These *noop*'s, as well as the instructions from the target times the replication factor, increase our code size in the same way as they do for the SPARC. The total code size increase when n=k+1 is shown in figure 8(d) with dashed lines. Overall, we see that the code size may increase by as little

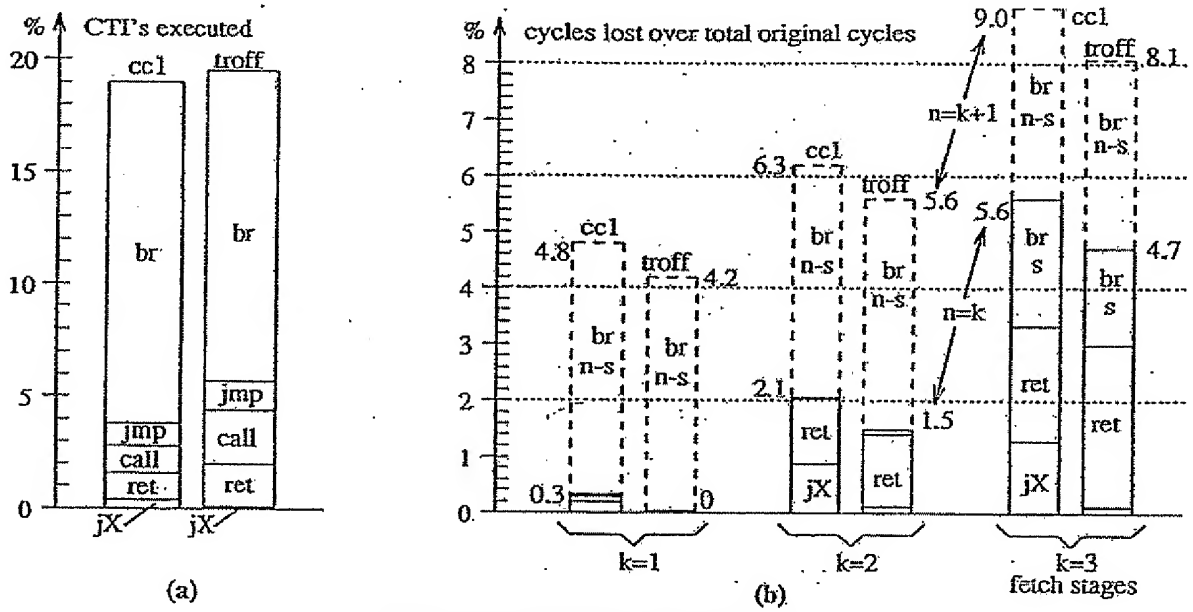


Figure 9: Dynamic measurements

Part (a) shows the percentage of the various CTI's that are executed in the original code, for the two benchmarks. Part (b) shows the additional execution cycles, over the total original cycles for executing the entire benchmark, when there are k fetch stages and when conditional branches have n delay slots. The additional cycles are broken down according to their main cause: non-selective branch slots, selective branch slots, delay slots of returns, delay slots of indexed jumps to *case*'s.

in this figure because their contribution to lost cycles is negligible: at most 0.001%. Procedure calls do not appear either because their delay slots are always filled, and thus no cycles are ever lost because of them. If, however, the linker did not copy instructions from the head of the called procedure into the delay slots of calls, then a major contribution to the lost cycles would be added: 0.3% (k=1), or 1.3% (k=2), or 2.5% (k=3) on the average.

The dashed lines in figure 9(b) are for the case n=k+1, when conditional branches have one more, non-selective delay slot. When that slot is filled with an instruction from the branch target and the branch fails, or when it is filled with a *noop*, then a cycle is lost; these cycles account for a major portion of the wasted cycles (about 4 % of the original cycles) when n=k+1. This number is exactly the same in our architecture as it would be in any conventional RISC with one branch delay slot; any compiler that can improve the utilization of that delay slot on a conventional RISC will also have the same effect on the scheme of this paper. In figure 9(b), the number of lost cycles is for SPARC code. Our corresponding measurements for MIPS code gave the same percentage of lost cycles for *cc1*, while *troff* gave a better (lower) percentage by about 1% for k=3.

The central point of this paper are the selective delay slots. Thus, it is worth looking more carefully to their contribution to wasted cycles; table 1 does that. This table contains fractions over the number of *executed branches* rather than the number of all executed instruction of figure 9. Thus, its numbers can be interpreted as "cycles per branch". The contribution of non-selective delay slots is ignored. The four contributions that may waste selective delay slot cycles are:

(i). *case* label inside branch delay slots (fig. 5(d));

(ii) branch followed by branch on the failure side (fig. 6(b));

(iii) branch followed by branch on the success side (fig. 6(a));

(iv) misalignment effect – see figure 7(a).

Table 1 shows the contribution of these factors.

Table 1: *Cycles per conditional branch (n=k)*

k fetch st.	*contributing factors*					Total
	br. itself	*(i)*	*(ii)*	*(iii)*	*(iv)*	
1	1.00	0.0007	0.0000	0.0000	0.0003	1.001
2	1.00	0.0014	0.0001	0.0215	0.0139	1.037
3	1.00	0.0022	0.0005	0.0880	0.0276	1.118

We see that the scheme of this paper dramatically decreases the number of cycles per conditional branch: even in the case k=3, when a dummy pipeline would take 4.0 cycles per branch, our scheme only takes 1.1 cycles per branch. The major contribution to wasted cycles is from branches closely following other branches on the success path.

This scheme requires somehow more code size expansion (e.g. 25 to 45 %), while it can reduce the cycles per branch down to 1.05 to 1.15 for two to four delay slots. Since the code size does not differ much, the main cost for the clearly better performance of the new scheme is the double-width I-fetch bus. Which way the tradeoff leans depends on the rest of the architecture and implementation. On-chip instruction RAM's can usually supply the additional bandwidth at small cost. Super-scalar architectures already have the additional bandwidth; for branches with little parallelism in their predicted target block, this scheme offers a good alternative for filling their delay slots.

ACKNOWLEDGEMENTS: Alain Lichnewsky and the anonymous reviewers gave us many insightful comments. Dennis Pnevmatikatos participated in the original discussions about rearranging instructions in a late-column-select memory and assisted us in those early stages. We thank them all.

REFERENCES

[ChHo87] P. Chow, M. Horowitz: "Architectural Tradeoffs in the Design of MIPS-X", Proceedings of the 14th Int. Symp. on Computer Architecture, ACM SIGARCH, 1987, pp. 300-308.

[DeRo87] J. DeRosa, H. Levy: "An Evaluation of Branch Architectures", Proceedings of the 14th Int. Symp. on Computer Architecture, ACM SIGARCH, 1987, pp. 10-16.

[Ditz87] D. Ditzel, H. McLellan: "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero" Proceedings of the 14th Int. Symp. on Computer Architecture, ACM SIGARCH, 1987, pp. 2-9.

[FaHe86] S. McFarling, J. Hennessy: "Reducing the Cost of Branches", Proceedings of the 13th Int. Symp. on Computer Architecture, ACM SIGARCH, 1986, pp. 396-403.

[GrHe82] T. Gross, J. Hennessy: "Optimizing Delayed Branches", Proceedings IEEE Micro-15, Oct. 1982, pp. 114-120.

[HwCC89] W. Hwu, T. Conte, P. Chang: "Comparing Software and Hardware Schemes for Reducing the Cost of Branches", Proceedings of the 16th Int. Symp. on Computer Architecture, ACM SIGARCH, 1989, pp. 224-233.

[JoWa89] N. Jouppi, D. Wall: "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Proceedings of the ASPLOS-III Conf., 1989 (ACM SIGARCH 17-2), pp. 272-282.

[LeSm84] J. Lee, A. Smith: "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer Magazine, v.17 n. 1, January 1984, pp. 6-22.

[Lilj88] D. Lilja: "Reducing the Branch Penalty in Pipelined Processors", IEEE Computer Magazine, v. 21 n. 7, July 1988, pp. 47-55.

[MIPS] J. Kane: "Mips RISC Architecture", Prentice Hall International, 1987.

[SPARC] see e.g. Cypress Semiconductor: "CY7C600 RISC family Users Guide – SPARC Architecture", June 1988.

[Wils89] P. Wilson: "The Architecture of the P1 – a 250 MHz SPARC in GaAs", presentation at the "Hot Chips Symposium", Stanford U., Palo Alto, California, 26 June 1989.